```
/*****************************************************************************
#   Copyright 1997 Silicon Graphics, Inc.  ALL RIGHTS RESERVED.
#
#   UNPUBLISHED -- Rights reserved under the copyright laws of the United
#   States.   Use of a copyright notice is precautionary only and does not
#   imply publication or disclosure.
#
#   THIS SOFTWARE CONTAINS CONFIDENTIAL AND PROPRIETARY INFORMATION OF
#   SILICON GRAPHICS, INC. ANY DUPLICATION, MODIFICATION, DISTRIBUTION, OR
#   DISCLOSURE IS STRICTLY PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN
#   PERMISSION OF SILICON GRAPHICS, INC.
#*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <strings.h>
#include <math.h>
#include "ri.h"
#include "ri_state.h"
#include <GL/gl.h>
#include "ri_shader.h"



/* temporary variable functions.  we maintain two circularly linked lists
   with active and free temporary variables.  when we need a new temp,
   we first see if we can grab one from the free list.  if so, we remove
   it from the free list and add it to the active list.  if not, we
   create a new temp and add it to the active list.  when a temp is freed,
   we move it from the active list to the free list.  */

static Temp *__active_temps = NULL;
static Temp *__free_temps = NULL;

Temp *new_temp(void)
{
    Temp *t;

    if( __free_temps ) {
      t = __free_temps;
      if( __free_temps->prev==__free_temps ) {
          __free_temps = NULL;
      } else {
          __free_temps->prev->next = __free_temps->next;
          __free_temps->next->prev = __free_temps->prev;
          __free_temps = __free_temps->prev;
      }
    } else {
       t = (Temp *)malloc(sizeof(Temp));
       glGenTexturesEXT(1,&t->id);
       glBindTextureEXT(GL_TEXTURE_2D,t->id);
       glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
       glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
       glBindTextureEXT(GL_TEXTURE_2D,0);
    }
```

```
    if( __active_temps ) {
      t->next = __active_temps;
      t->prev = __active_temps->prev;
        __active_temps->prev->next = t;
        __active_temps->prev = t;
    } else {
      __active_temps = t;
      __active_temps->prev = t;
      __active_temps->next = t;
    }

    return t;
}

void free_temp(Temp *t)
{
    /* XXX assumes temp really is on active list! */
    if( t->prev==t ) {
      __active_temps = NULL;
    } else {
        t->prev->next = t->next;
        t->next->prev = t->prev;
      if( t==__active_temps ) {
            __active_temps = t->prev;
      }
    }

    if( __free_temps ) {
        t->next = __free_temps;
        t->prev = __free_temps->prev;
        __free_temps->prev->next = t;
        __free_temps->prev = t;
    } else {
        __free_temps = t;
        __free_temps->prev = t;
        __free_temps->next = t;
    }
}

void free_temps(void)
{
    Temp *t, *tt;

    if( __active_temps==NULL ) {
      return;
    }

    t = __active_temps->next;
    while( t!=__active_temps ) {
      tt = t->next;
      free_temp(t);
      t = tt;
    }
    free_temp(__active_temps);

/*
```

```
    if( __free_temps==NULL ) {
      __free_temps = __active_temps;
    } else {
        __active_temps->prev->next = __free_temps;
        __free_temps->prev = __active_temps->prev;
        __active_temps->next->prev = __free_temps->prev->next;
        __free_temps->prev->next = __active_temps->next;
    }
*/

    __active_temps = NULL;
}



/* initialization and cleanup routines for the shader being executed.  we
   store the original underlying pixels in blend_temp in case we must
   blend a partially transparent result into the framebuffer.  on cleanup,
   we restore these pixels by writing over the new image that has been
   created by this shader. */

FILE *__shader_open(char *name)
{
    char path[1024];
    char n[256];
    char *c, *cp;
    FILE *fp;

    strcpy(path,getenv("SHADERS"));

    cp = path;
    c = strchr(cp,':');

    while( c!=NULL ) {
      *c = '\0';
        strcpy(n,cp);
        strcat(n,"/");
        strcat(n,name);
        strcat(n,".soo");
        if( (fp = fopen(n,"r"))!=NULL ) {
          return fp;
      }
      cp = c+1;
      c = strchr(cp,':');
    }

    strcpy(n,cp);
    strcat(n,"/");
    strcat(n,name);
    strcat(n,".soo");

    return fopen(n,"r");
}

Shader *__shader_install(char *name, Shader **list,
    void *(*shader)(char *name, RtInt n, RtToken tokens[], RtPointer values[]))
```

```
{
    Shader *s;

    s = (Shader *)malloc(sizeof(Shader));
    s->name = (char *)malloc(strlen(name)+1);
    strcpy(s->name,name);
    s->shader = shader;
    s->next = *list;
    s->L = NULL;
    s->Cl = NULL;
    *list = s;

    return(s);
}

extern void *pack_args(char *nm, RtInt n, RtToken tokens[], RtPointer
values[]);

Shader *__shader_lookup(char *name, Shader **list)
{
    Shader *s;
    FILE *fp;

    s = *list;
    while( s!=NULL ) {
        if( !strcmp(s->name,name) )
            return(s);
        s = s->next;
    }

    fp = __shader_open(name);
    if( fp==NULL ) {
        fprintf(stderr,"no surface shader file:  %s\n",name);
        return NULL;
    }

    /* pack args here? */

    fclose(fp);

    s = __shader_install(name,list,pack_args);

    return s;
}



static Temp *__blend_temp = NULL;

void __shader_init(RiAttributes *att, Dlist *dlist)
{
    DrawOp d;

    __blend_temp = new_temp();
    __reg_store(__blend_temp,rgba_rgba);

    /* set alpha to zero to do looping */
```

```
    glColorMask(0,0,0,1);
    d.cscale[0] = 1.;
    d.cscale[1] = 1.;
    d.cscale[2] = 1.;
    d.cscale[3] = 0.;
    d.op = __ps_flatpoly;
    __fb_load(&d);
    glColorMask(1,1,1,1);

    /* lay in stencil image to mask where we have geometry */
    d.dlop = dlist->list;
    d.att = att;
    d.cscale[0] = 1.;
    d.cscale[1] = 1.;
    d.cscale[2] = 1.;
    d.cscale[3] = 1.;
    d.op = __ps_geometry;

    glClear( GL_STENCIL_BUFFER_BIT );
    glStencilFunc(GL_ALWAYS, 0x1, 0x1);
    glStencilOp(GL_REPLACE, GL_KEEP, GL_REPLACE);
    glEnable(GL_STENCIL_TEST);

    __fb_load(&d);

    glStencilFunc(GL_EQUAL, 0x1, 0x1);
    glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
}

void __shader_cleanup(void)
{
    if( __blend_temp ) {
        DrawOp drop;

        glEnable(GL_BLEND);
        glBlendFunc(GL_ONE_MINUS_DST_ALPHA,GL_DST_ALPHA);

        drop.cscale[0] = 1.;
        drop.cscale[1] = 1.;
        drop.cscale[2] = 1.;
        drop.cscale[3] = 1.;
        drop.op = __ps_texpoly;
        drop.temp = __blend_temp;
        drop.lut = drop.temp->id;
        __fb_load(&drop);

        glDisable(GL_BLEND);
        glBlendFunc(GL_ONE,GL_ZERO);

        glDisable(GL_STENCIL_TEST);

        free_temp(__blend_temp);
        __blend_temp = NULL;
    } else {
        glDisable(GL_STENCIL_TEST);
    }
```

```
    free_temps();
}









/* antiquated functions; may put back in or not in the future */

void __sp_normaleye(DrawOp *drop)
{
    DlistOp *dlop = drop->dlop;
    RiAttributes *att = drop->att;
    /* float *col = drop->cscale; */
    float whi[4] = {1.,1.,1.,1.};
    float blk[4] = {0.,0.,0.,1.};
    float x[4] = {1.,0.,0.,0.};
    float y[4] = {0.,1.,0.,0.};
    float z[4] = {0.,0.,1.,0.};

    __material_set(blk,whi,blk,1.);

    glPushMatrix();
    glLoadIdentity();
    glLightfv(GL_LIGHT1, GL_AMBIENT, blk);
    glLightfv(GL_LIGHT1, GL_DIFFUSE, x);
    glLightfv(GL_LIGHT1, GL_SPECULAR, blk);
    glLightfv(GL_LIGHT1, GL_POSITION, x);
    glLightfv(GL_LIGHT2, GL_AMBIENT, blk);
    glLightfv(GL_LIGHT2, GL_DIFFUSE, y);
    glLightfv(GL_LIGHT2, GL_SPECULAR, blk);
    glLightfv(GL_LIGHT2, GL_POSITION, y);
    glLightfv(GL_LIGHT3, GL_AMBIENT, blk);
    glLightfv(GL_LIGHT3, GL_DIFFUSE, z);
    glLightfv(GL_LIGHT3, GL_SPECULAR, blk);
    glLightfv(GL_LIGHT3, GL_POSITION, z);
    glPopMatrix();

    glEnable(GL_LIGHTING);
    glLightModelf(GL_LIGHT_MODEL_TWO_SIDE,GL_FALSE);

    glDisable(GL_LIGHT0);
    glEnable(GL_LIGHT1);
    glEnable(GL_LIGHT2);
    glEnable(GL_LIGHT3);

    __ri_setattributes(att,NULL);
    dlist_execute(dlop);

    glPushMatrix();
    glLoadIdentity();
```

```
    x[0] = -1.;
    glLightfv(GL_LIGHT1, GL_POSITION, x);
    y[1] = -1.;
    glLightfv(GL_LIGHT2, GL_POSITION, y);
    z[2] = -1.;
    glLightfv(GL_LIGHT3, GL_POSITION, z);
    glPopMatrix();

    glEnable(GL_BLEND);
    glBlendFunc(GL_ONE, GL_ONE);
    glBlendEquationEXT(GL_FUNC_REVERSE_SUBTRACT_EXT);
    __ri_setattributes(att,NULL);
    dlist_execute(dlop);
    glBlendEquationEXT(GL_FUNC_ADD_EXT);
    glBlendFunc(GL_ONE, GL_ZERO);
    glDisable(GL_BLEND);

    glEnable(GL_LIGHT0);
    glDisable(GL_LIGHT1);
    glDisable(GL_LIGHT2);
    glDisable(GL_LIGHT3);

    glLightModelf(GL_LIGHT_MODEL_TWO_SIDE,GL_TRUE);
    glFragmentLightModelfSGIX(GL_FRAGMENT_LIGHT_MODEL_TWO_SIDE_SGIX,GL_TRUE);
    glDisable(GL_LIGHTING);
}

void __sl_n(DrawOp *drop)
{
    DlistOp *dlop = drop->dlop;
    RiAttributes *att = drop->att;
    float *col = drop->cscale;
    float blk[4] = {0.,0.,0.,1.};
    float x[4] = {1.,0.,0.,0.};
    float y[4] = {0.,1.,0.,0.};
    float z[4] = {0.,0.,1.,0.};

    __material_set(blk,col,blk,1.);

    glPushMatrix();
    glLoadMatrixf((GLfloat *)CurOptions->worldtocamera);
    glLightfv(GL_LIGHT1, GL_AMBIENT, blk);
    glLightfv(GL_LIGHT1, GL_DIFFUSE, x);
    glLightfv(GL_LIGHT1, GL_SPECULAR, blk);
    glLightfv(GL_LIGHT1, GL_POSITION, x);
    glLightfv(GL_LIGHT2, GL_AMBIENT, blk);
    glLightfv(GL_LIGHT2, GL_DIFFUSE, y);
    glLightfv(GL_LIGHT2, GL_SPECULAR, blk);
    glLightfv(GL_LIGHT2, GL_POSITION, y);
    glLightfv(GL_LIGHT3, GL_AMBIENT, blk);
    glLightfv(GL_LIGHT3, GL_DIFFUSE, z);
    glLightfv(GL_LIGHT3, GL_SPECULAR, blk);
    glLightfv(GL_LIGHT3, GL_POSITION, z);
    glPopMatrix();

    /* __light_enable(); */
    glEnable(GL_LIGHTING);
```

```
    glLightModelf(GL_LIGHT_MODEL_TWO_SIDE,GL_FALSE);
    glFragmentLightModelfSGIX(GL_FRAGMENT_LIGHT_MODEL_TWO_SIDE_SGIX,GL_FALSE);

    glDisable(GL_LIGHT0);
    glEnable(GL_LIGHT1);
    glEnable(GL_LIGHT2);
    glEnable(GL_LIGHT3);

    __ri_setattributes(att,NULL);
    dlist_execute(dlop);

    glPushMatrix();
    glLoadMatrixf((GLfloat *)CurOptions->worldtocamera);
    x[0] = -1.;
    glLightfv(GL_LIGHT1, GL_POSITION, x);
    y[1] = -1.;
    glLightfv(GL_LIGHT2, GL_POSITION, y);
    z[2] = -1.;
    glLightfv(GL_LIGHT3, GL_POSITION, z);
    glPopMatrix();

    glEnable(GL_BLEND);
    glBlendFunc(GL_ONE, GL_ONE);
    glBlendEquationEXT(GL_FUNC_REVERSE_SUBTRACT_EXT);
    __ri_setattributes(att,NULL);
    dlist_execute(dlop);
    glBlendEquationEXT(GL_FUNC_ADD_EXT);
    glBlendFunc(GL_ONE, GL_ZERO);
    glDisable(GL_BLEND);

    glEnable(GL_LIGHT0);
    glDisable(GL_LIGHT1);
    glDisable(GL_LIGHT2);
    glDisable(GL_LIGHT3);

    glLightModelf(GL_LIGHT_MODEL_TWO_SIDE,GL_TRUE);
    glFragmentLightModelfSGIX(GL_FRAGMENT_LIGHT_MODEL_TWO_SIDE_SGIX,GL_TRUE);
    glDisable(GL_LIGHTING);
    /* __light_disable(); */
}

void __sl_ndotv(DrawOp *drop)
{
    DlistOp *dlop = drop->dlop;
    RiAttributes *att = drop->att;
    float *col = drop->cscale;
    float blk[4] = {0.,0.,0.};
    float whi[4] = {1.,1.,1.};
    float zer[4] = {0.,0.,1.};

    __material_set(blk,col,blk,30.);

    glPushMatrix();
    glLoadMatrixf((GLfloat *)CurOptions->worldtocamera);
    glLightfv(GL_LIGHT1, GL_AMBIENT, blk);
    glLightfv(GL_LIGHT1, GL_DIFFUSE, whi);
    glLightfv(GL_LIGHT1, GL_SPECULAR, blk);
```

```
    glLightfv(GL_LIGHT1, GL_POSITION, zer);
    glPopMatrix();

    /* __light_enable(); */
    glEnable(GL_LIGHTING);

    glDisable(GL_LIGHT0);
    glEnable(GL_LIGHT1);

    __ri_setattributes(att,NULL);
    dlist_execute(dlop);

    glEnable(GL_LIGHT0);
    glDisable(GL_LIGHT1);

    glDisable(GL_LIGHTING);
    /* __light_disable(); */
}
```